Representing Block-structured Process Models as Order Matrices: Basic Concepts, Formal Properties, Algorithms

Chen Li[1] ⋆, Manfred Reichert[2], and Andreas Wombacher[3]

[1] Information Systems Group, University of Twente, The Netherlands
lic@cs.utwente.nl
[2] Institute of Databases and Information Systems, Ulm University, Germany
manfred.reichert@uni-ulm.de
[3] Database Group, University of Twente, The Netherlands
a.wombacher@utwente.nl

Abstract. In various cases we need to transform a process model into a matrix representation for further analysis. In this paper, we introduce the notion of Order Matrix, which enables unique representation of block-structured process models. We present algorithms for transforming a block-structured process model into a corresponding order matrix and vice verse. We then prove that such order matrix constitutes a unique representation of a block-structured process model; i.e., if we transform a process model into an order matrix, and then transform this matrix back into a process model, the two process models are trace equivalent; i.e., they show same behavior. Finally, we analyze algebraic properties of order matrices.

1 Introduction

In various cases we need to transform a process model into a matrix representation for further analysis. For example, in graph theory *adjacency matrices* are often used for various kinds of graph analysis (e.g., reachability analysis or derivation of minimal spanning tree [25]). In process mining, *causal matrices* are used to represent the relationship between transitions in Petri nets. Causal matrices are further applied in genetic process mining to discover a process model which covers the execution traces of a collection of process instances best [7]. In the field of data mining, matrices are used to classify, cluster or associate data [26, 20]. However, all these techniques are focusing on the nodes and edges of a graph or process model, and cannot be applied in respect to the management of process changes [12].

In this paper, we introduce the notion of *order matrix*, which represents all transitive relations between the activities of a block-structured process model. In the context of managing process variants [8], for example, we have already

⋆ This work was done in the MinAdept project, which has been supported by the Netherlands Organization for Scientific Research under contract number 612.066.512.

applied this kind of matrix for measuring the structural similarity between two process models [13]. We have further used order matrices for mining structurally different process variants. Here, we aim at discovering a process model that structurally covers a collection of process variants best [11, 14]. The present paper focuses on basic concepts, algorithms and formal properties of order matrices, and less on their use.

The remainder of this paper is organized as follows: Section 2 introduces some basic definitions needed for understanding this paper. Section 3 then provides the formal definition of an order matrix and gives an illustrative example. Section 4 presents an algorithm for transforming a block-structured process model into an order matrix. Section 5 presents an algorithm for transforming an order matrix back into a block-structured process model. In Section 6, we prove that there exists a one-to-one mapping between a process model and its order matrix, i.e., if one transforms a process model into an order matrix, and then transform this matrix back into a process model, the two models will be same. Finally, we present algebraic properties of order matrices in Section 7.

2 Backgrounds

We first introduce basic notions needed in the following:

Process Model: Let $\mathcal{P}$ denote the set of all sound (i.e., correct) process models. We denote a process model as sound if there are no deadlocks or unreachable activities in the process model [21, 28]. In our context, a particular *process model* $S = (N, E, \ldots)^4 \in \mathcal{P}$ is defined in terms of an Activity Net [21]. N constitutes the set of activities and E the set of control edges (i.e., precedence relations) linking them. More precisely, Activity Nets cover the following fundamental process patterns: Sequence, AND-split, AND-join, XOR-split, XOR-join, and Loop [27].[5] These patterns constitute the core set of any workflow specification language (e.g., WS-BPEL [3] and BPMN [4]) and cover most of the process models we can find in practice [36, 15]. Furthermore, based on these patterns we are able to compose more complex ones if required (e.g., an OR-split can be mapped to XOR- and AND- splits [19]). Finally, when restricting process modeling to these basic process patterns, we obtain models that are better understandable and less erroneous [18, 16]. A simple example of an Activity Net is depicted in Fig. 1a. For a detailed description and correctness issues we refer to [21].

Block Structuring: To limit the scope, we assume Activity Nets to be block-structured, i.e., sequences, branchings (with aforementioned split and join semantics), and loops are specified as blocks with well-defined start and end nodes. These blocks may be nested, but must not overlap, i.e., the nesting must

[4] A Well-structured Activity Net contains more elements than only node set N and edge set E, which can be ignored in the context of this paper.

[5] These patterns can be mapped to other languages as well. For example in Business Process Execution Language (BPEL), XOR-Split / -join can be represented by 'If' or 'Pick', AND-Split / -Join by 'Flow', and Loops by 'While' or 'RepeatUntil' [3].

be regular [21, 10]. A block in a process model S can be a single activity, a self-contained part of S, or even S itself. As example consider process model S from Fig. 1. Here {A}, {A,B}, {C,F}, and {A,B,C,D,E,F,G} describe possible blocks contained in S. Note that we can represent a block B as activity set, since the block structure itself already becomes clear from the process model S. For example, block {A,B} corresponds to the parallel block with corresponding AND-split and AND-join nodes in S. The concept of block-structuring can be found in languages like WS-BPEL [3]. When compared with non-block-structured process models, block-structured ones are easier understandable for users and have less chances of containing errors [23, 16–18]. If a process model is not block-structured, in most practically relevant cases we can transform it into a block-structured one (see [31, 18, 10]).

Definition 1 (Trace). *Let $S = (N, E, \ldots) \in \mathcal{P}$ be a process model. We define t as a trace of S iff:*

- *$t \equiv < a_1, a_2, \ldots, a_k >$ (with $a_i \in N$) constitutes a valid and complete execution sequence of activities considering the control flow defined by S. We define $\mathcal{T}_S$ as the set of all traces that can be produced by process instances running on process model S.*
- *$t(a \prec b)$ is denoted as precedence relationship between activities a and b in trace $t \equiv < a_1, a_2, \ldots, a_k >$ iff $\exists i < j : a_i = a \wedge a_j = b$.*

We only consider traces composing 'real' activities, but no events related to silent activities, i.e., nodes in a process model having no associated action and only existing for control flow purpose [13]. At this stage, we consider two process models as being the same if they are *trace equivalent*, i.e., $S \equiv S'$ iff $\mathcal{T}_S \equiv \mathcal{T}_{S'}$. Like most process mining approaches [30, 7, 34], the stronger notion of bi-similarity [9] is not considered in our context.

3 Basic Definition of an Order Matrix

One key feature of our ADEPT change framework is to maintain the structure of the unchanged parts of a process model [21, 6, 33]. For example, when deleting an activity this neither influences successors nor predecessors of this activity, and therefore also not their order relations [24, 22]. To incorporate this feature in our approach, rather than only looking at direct predecessor-successor relationships between activities (i.e. control edges), we consider the transitive control dependencies for each pair of activities; i.e., for a given process model $S = (N, E, \ldots) \in \mathcal{P}$, for activities $a_i, a_j \in N$, $a_i \neq a_j$ we examine their structural order relations (including transitive order relations). Logically, we determine order relations by considering all traces in trace set $\mathcal{T}_S$ producible by model S.

Fig. 1a shows an example of a process model S. This model comprises process patterns like Sequence, AND-block, XOR-block, and Loop-block [27]. Here, trace set $\mathcal{T}_S$ of S constitutes an infinite set due to the presence of the loop-block in S (cf. Fig. 1b). Such infinite number of traces precludes us to perform any detailed analysis of the trace set. Therefore we need to transform such infinite trace sets into a finite representation for further analysis.

3.1 Simplification of Infinite Trace Sets

One common approach to represent a string with infinite length is to represent it as finite set of n-gram lists [5]. The general idea behind an n-gram list is to represent a single string by an ordered list of substrings with length n (so-called n-grams). In particular, only the first occurrence of an n-gram is considered, while later occurrences of the same n-gram are omitted in the n-gram list. Thus, an n-gram list represents a collection of strings with different length. In particular, an infinite language can be represented as finite set of n-gram lists. For example, a string $< abababab >$ can be represented as 2-gram $< \$a, ab, ba, b\# >$, where \$ (#) represents the start (end) of the string. Such approach is commonly used for analyzing loop structures in process models [35, 2], or - more generally - in the context of text indexing for substring matching [1]. Inspired by the n-gram approach, we define the notion of *Simplified Trace Set* as follows:

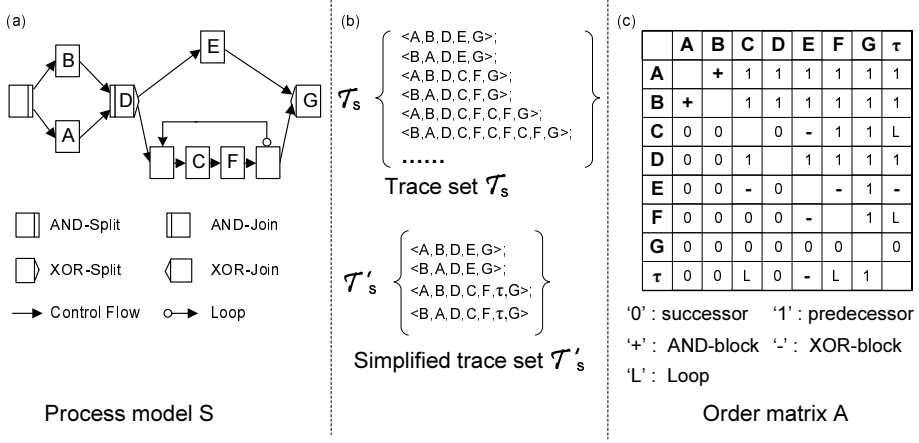
Definition 2 (Simplified Trace Set).
Let S be a process model and $\mathcal{T}_S$ denote the trace set producible on S. Let B_k, $k = (1, \ldots, K)$ be Loop-blocks in S, and $\mathcal{T}_{B_k}$ denote the set of traces producible on B_k. Let further $(t_{B_k})^m$ be a sequence of m ($m \in \mathbb{N}$) traces $< t^1_{B_k}, t^2_{B_k}, \ldots, t^m_{B_k} >$ with $t^j_{B_k} \in \mathcal{T}_{B_k}$, $j \in \{1, \ldots, m\}$. We additionally define $(t_{B_k})^0 =<>$ as an empty sequence. If we only consider the activities corresponding to B_k, in any trace $t \in \mathcal{T}_S$ producible on S, t either has no entries [6] or must have structure $< t^*_{B_k}, (t_{B_k})^m >$, with $t^*_{B_k} \in \mathcal{T}_{B_k}$ representing the first loop iteration and $m \in \mathbb{N}_0$ being the number of additional iterations loop-block B_k is executed in trace t. We can simplify this structure by using $< t_{B_k}, \tau_k >$ instead, where τ_k refers to $(t_{B_k})^m$. When simplifying trace set $\mathcal{T}_S$ this way, we obtain a finite set of traces $\mathcal{T}'_S$, denoted as `Simplified Trace Set` of process model S.

In our representation of a trace $t \in \mathcal{T}_S$, we only consider the first occurrence of trace $t^*_{B_k}$ producible by block B_k while omitting others that occur later within trace t. Instead, we represent such repetitive entries by a silent activity τ_k, which has not associated action but solely exists to indicate omission of other t_{B_k} appearing later in trace t, i.e., τ_k represents the iterative execution of loop-block B_k captured in trace t.[7] When omitting repetitive entries within trace set $\mathcal{T}_S$, we obtain a finite trace set $\mathcal{T}'_S$ that we can use for further analysis. Note that when dealing with nested loops (e.g., a loop-block B_k contains another loop-block B_j), we first need to analyze B_j and then B_k; i.e., we need to first define τ_j to represent the iterative execution of loop-block B_j as captured in trace t and then define τ_k to represent loop-block B_k.

As example consider process model S in Fig. 1a. Loop-block B, which is surrounded by a loop-backward edge, is the block comprising activities `C` and

[6] i.e., the loop-block B_k has not been executed at all.

[7] Though this approach is inspired by n-gram, it is different from n-gram representation of a string. In n-gram the length of the sub-string is a fixed number n, while in our approach we use τ_k to represent traces producible by the Loop-block B_k. Obviously, traces producible by B_k do not necessarily have same length.



	A	B	C	D	E	F	G	τ
A		+	1	1	1	1	1	1
B	+		1	1	1	1	1	1
C	0	0		0	-	1	1	L
D	0	0	1		1	1	1	1
E	0	0	-	0		-	1	-
F	0	0	0	0	-		1	L
G	0	0	0	0	0	0		0
τ	0	0	L	0	-	L	1	

Fig. 1. a) Process model and b) related order matrix

F; consequently the trace set this block can produce is $\{< \texttt{C},\texttt{F} >\}$. Therefore, any trace $t \in \mathcal{T}_S$ producible on S has structure $< \texttt{C},\texttt{F},(\texttt{C},\texttt{F})^m >$ with $m \in \mathbb{N}_0$ depending on the number of times the loop iterates. For example, $< \texttt{C},\texttt{F} >$, $< \texttt{C},\texttt{F},\texttt{C},\texttt{F} >$ and $< \texttt{C},\texttt{F},\texttt{C},\texttt{F},\texttt{C},\texttt{F} >$ are all valid traces producible by the loop-block. Let us define a silent activity τ corresponding to block B. Then we can simplify these traces by $< \texttt{C},\texttt{F},\tau >$ where τ refers the to the sequence of the traces producible on B. This way, we can simplify infinite trace set $\mathcal{T}_S$ to finite set $\mathcal{T}'_S = \{< \texttt{A},\texttt{B},\texttt{D},\texttt{E},\texttt{G} >, < \texttt{B},\texttt{A},\texttt{D},\texttt{E},\texttt{G} >, < \texttt{A},\texttt{B},\texttt{D},\texttt{C},\texttt{F},\tau,\texttt{G} >, < \texttt{B},\texttt{A},\texttt{D},\texttt{C},\texttt{F},\tau,\texttt{G} >\}$ (cf. Fig. 1b).

3.2 Defining an Order Matrix

For process model S, the analyzing results of its trace set $\mathcal{T}_S$ are aggregated in an order matrix A, which considers five types of order relations (cf. Def. 3):

Definition 3 (Order matrix). *Let $S = (N, E, \ldots) \in \mathcal{P}$ be a process model with activity set $N = \{a_1, a_2, \ldots, a_n\}$. Let further $\mathcal{T}_S$ denote the set of all traces producible on S and let $\mathcal{T}'_S$ be the simplified trace set of S (cf. Def. 2). Let B_k, $k = (1, \ldots, K)$ denote loop-blocks in S. For every B_k, we define silent activity τ_k, $k = (1, \ldots, K)$ to represent the iterative structure producible by B_k in $\mathcal{T}'_S$. Then:*

*A is called **order matrix** of S with $A_{a_i a_j}$ representing the order relation between activities a_i,$a_j \in N \bigcup \{t_k \big| k = 1, \ldots, K\}$, $i \neq j$ iff:*

- *$A_{a_i a_j}$ = '1' iff ($\forall t \in \mathcal{T}'_S$ with $a_i, a_j \in t \Rightarrow t(a_i \prec a_j)$)*
  If for all traces containing activities a_i and a_j, a_i always appears BEFORE *a_j, we denote $A_{a_i a_j}$ as '**1**', i.e., a_i always precedes of a_j in the flow of control.*
- *$A_{a_i a_j}$ = '0' iff ($\forall t \in \mathcal{T}'_S$ with $a_i, a_j \in t \Rightarrow t(a_j \prec a_i)$)*

If for all traces containing activities a_i and a_j, a_i always appears AFTER *a_j, we denote $A_{a_i a_j}$ as a '**0**', i.e. a_i always succeeds of a_j in the flow of control.*

- $A_{a_i a_j}$ = '+' *iff* $(\exists t_1 \in \mathcal{T}'_{\mathcal{S}}$, *with* $a_i, a_j \in t_1 \wedge t_1(a_i \prec a_j)) \wedge (\exists t_2 \in \mathcal{T}'_{\mathcal{S}}$, *with* $a_i, a_j \in t_2 \wedge t_2(a_j \prec a_i))$
  If there exists at least one trace in which a_i appears before a_j and another trace in which a_i appears after a_j, we denote $A_{a_i a_j}$ as '+', i.e. a_i and a_j are contained in different parallel branches.
- $A_{a_i a_j}$ = '-' *iff* $(\neg\exists t \in \mathcal{T}'_{\mathcal{S}} : a_i \in t \wedge a_j \in t)$
  If there is no trace containing both activity a_i and a_j, we denote $A_{a_i a_j}$ as '-', i.e. a_i and a_j are contained in different branches of a conditional branching.
- $A_{a_i a_j}$ = 'L', *iff* $((a_i \in B_k \wedge a_j = \tau_k) \vee (a_j \in B_k \wedge a_i = \tau_k))$
  *For any activity a_i in a Loop-block B_k, we define order relation $A_{a_i \tau_k}$ between it and the corresponding silent activity τ_k as '**L**'.*

The first four order relations {1,0,+,-} specify the precedence relations between activities as captured in the trace set, while the last order relation 'L' indicates loop structures within the trace set. Fig. 1c presents the order matrix of process model S. Since S contains one Loop-block, a silent activity τ is also added to this matrix. This order matrix contains all five order relations as described in Definition 3. For example, activities `E` and `C` will never appear in same trace belonging to the simplified trace set since they are contained in different branches of an XOR block. Therefore, we assign '-' to matrix element $A_{\mathtt{EC}}$. Further, since in all traces which contain both activities `B` and `G`, `B` always appears before `G`, we can obtain order relation $A_{\mathtt{BG}}$ = '1' and order relation $A_{\mathtt{GB}}$ = '0'. Special attention should be paid to the order relations between the silent activity τ and the other activities. The order relation between τ and activities `C` and `F` is set to 'L', since both `C` and `F` are contained in the Loop-block; for all remaining activities, τ has same order relations with them as activities `C` or `F` have. Note that the main diagonal of the order matrix is empty, since we do not compare an activity with itself.

As one can see, elements $A_{a_i a_j}$ and $A_{a_j a_i}$ can be derived from each other. If activity a_i is a predecessor of activity a_j, (i.e. $A_{a_i a_j} = 1$), we can always conclude that $A_{a_j a_i} = 0$ holds and if $A_{a_i a_j} \in \{$'+','-', 'L'$\}$, we obtain $A_{a_j a_i} = A_{a_i a_j}$.

4 Transforming a Process Model into an Order Matrix

Clearly, it is not realistic to first enumerate all traces of a process model and analyze the order relation based on them. The trace set of a process model can be extremely large particularly if the model contains several AND-blocks or even infinite if there are loop-blocks. In the following, we introduce Algorithm 1 to compute the order matrix for a process model in polynomial time. Note that this algorithm is also able to cope with loop structures.

In Algorithm 1, we first define set $P(a_i)$ for each activity $a_i \in N$, which contains all (direct and indirect) predecessors of a_i (Line 1). An activity a_j is

input : A process model $S = (N, E, \ldots)$
output: Its order matrix A

For each activity $a_i \in N, i = (1, \ldots, n)$, define $P(a_i)$ as the predecessor set of a_i ;
Define Set C as the set of activities which are already parsed;
Define $\mathcal{L}$ as the set of Loop-blocks B_k;
$P(a_i) = \emptyset$ for $i = (1, \ldots, n)$, $\mathsf{C} = \emptyset$ and $L = \emptyset$ `/* initial state */`;
Set `parseModel` (Node t_{start}, Node t_{end}, Set C) **begin**
    `/* Compute predecessor sets for activities between node` t_{start} `and` t_{end}`. Returns a new C' */`
    while *($t_{start} \neq t_{end}$)* **do**
        if *(Sequence)* **then**
            $P(t_{start}) = P(t_{start})$.`addAll`($\mathsf{C}$) ;
            $\mathsf{C}' = \mathsf{C}$.`add`(t_{start}) ;
            $t_{start} = t_{start}$.`nextNode`;
        else if *(XOR-block)* **then**
            foreach *branch i in XOR-split,* $i = (1, \ldots, m)$ **do**
                n_i = XOR-split.`nextNode` in branch i;
                C'_i = `parseModel` (n_i, XOR-join, C) ;
            $\mathsf{C}' = \mathsf{C}$.`addAll`($\bigcup_{i=1}^{m} \mathsf{C}'_i$) ;
            t_{start} = XOR-join.`nextNode`;
        else if *(AND-block)* **then**
            foreach *branch i in AND-split,* $i = (1, \ldots, m)$ **do**
                n_i = XOR-split.`nextNode` in branch i;
                C'_i = `parseModel` (n_i, XOR-join, C) ;
            foreach *branch k in AND-split,* $k = (1, \ldots, m)$ **do**
                $\mathsf{C}_k = \bigcup_{i=1, i \neq k}^{m} \mathsf{C}'_i$;
                `parseModel` (AND-split, AND-join, C_k);
            $\mathsf{C}' = \mathsf{C}$.`addAll`($\bigcup_{i=1}^{m} \mathsf{C}'_i$) ;
            t_{start} = AND-join.`nextNode`;
        else if *(Loop-block)* **then**
            $\mathcal{L}$.`add` (`parseModel` (Loop-start.`nextNode`, Loop-end, $\emptyset$)) ;
            C' = C.`addAll` (`parseModel` (Loop-start,Loop-end, C)) ;
            t_{start} = Loop-end.`nextNode`;
    `return` C' ;
end
`computeOrderRelationBasedOnPredecessorSets` () **begin**
    `/* Compute order relation based on predecessor sets */` ;
    foreach $a_i, a_j \in N, i \neq j$ **do**
        if $P(a_i)$`contain`$(a_j) \wedge \neg P(a_i)$`contain`(a_j) **then** A_{ij} = '1';
        else if $\neg P(a_i)$`contain`$(a_j) \wedge P(a_i)$`contain`(a_j) **then** A_{ij} = '0';
        else if $P(a_i)$`contain`$(a_j) \wedge P(a_i)$`contain`(a_j) **then** A_{ij} = '+';
        else if $\neg P(a_i)$`contain`$(a_j) \wedge \neg P(a_i)$`contain`(a_j) **then** A_{ij} = '-';
end
`addSilentActivitiesForLoopStructure` (Set $\mathcal{L}$, OrderMatrix A) **begin**
    `/* Add loop on order matrix */` ;
    foreach $B_k \in \mathcal{L}$ **do**
        Define silent activity τ_k; $N = N' \bigcup \{\tau_k\}$;
        foreach $a_i \in N'$ **do**
            if $(a_i \in B_k)$ **then**
                $A_{a_i \tau_k}$ = 'l' ; $A_{\tau_k a_i}$ = 'l' ;
            else if *($a_i \in N' \setminus L_i$)* **then**
                Let $a_j \in L_i$;
                $A_{a_i \tau_k} = A_{a_i a_j}$; $A_{\tau_k a_i} = A_{a_j a_i}$;
end

Algorithm 1: Computing order matrix for process model

a predecessor of a_i iff $\exists t \in \mathcal{T}_S' : t(a_j \prec a_i)$. In addition, we define set $\mathcal{L}$ which contains all Loop-blocks B_k of the process model. Following that, three functions are specified. Function `parseModel` (Lines 5 to 31) first parses the process model S and computes the predecessor sets $P(a_i)$ for each activity of the model. Further, it determines set $\mathcal{L}$ which contains all Loop-blocks B_k of S. Then function `computeOrderRelationBasedOnPredecessorSets` (lines 32 to 38) calculates the order relations for every pair of activities of model S based on their predecessor sets. Finally, function `addSilentActivitiesForLoopStructure` (lines 39 to 48) specially handles the loop structures of process model S.

Function `parseModel` has three input parameters: Nodes t_{start} and t_{end} mark the start and the end of the block B_i we need to analyze; set `C` corresponds to the set of activities already been analyzed. After computing predecessor sets and Loop-blocks for the activities from block B_i, we obtain new set `C'` comprising original set `C` and all activities of B_i. Initially, t_{start} is set to the start-flow of S, t_{end} to the end-flow of S, and `C` to an empty set (Line 5).

In our analysis, we consider four process patterns:

- **Sequence**. This pattern is analyzed in Lines 7 to 10 in Algorithm 1. Assume that blocks B_i, B_j and B_k are three blocks of process model S, where B_i precedes B_j and B_j precedes B_k. Let $a_i \in B_i$ and $a_j \in B_j$. For any trace $t \in \mathcal{T}_S$ containing both a_i and a_j, we obtain $t(a_i \prec a_j)$. Therefore, for every $a_j \in B_j$, we need to add B_i to its predecessor set $P(a_j)$. Similarly, for every $a_k \in B_k$, we need to add both B_i and B_j to its predecessor set $P(a_k)$. In Algorithm 1, we first add B_i to set `C`, and then add `C` to every $P(a_j) \in B_j$. Finally we add B_j to `C` (lines 7-9). We repeat same procedure when analyzing B_k, i.e., we add `C` to every $P(a_k)$ with $a_k \in B_k$, and then add B_k to `C`.
- **XOR-block**. This pattern is analyzed in Lines 10 - 15 in Algorithm 1. Since the activities in one branch of an XOR-block can never appear in trace t together with activities of another branch of same XOR-block, we analyze each branch of an XOR-block separately. Every branch i is considered as block of S, which can be analyzed independently by the `parseModel` function. In this case, t_{start} shall point to the first node on branch i and t_{end} to the XOR-join node. Further we need to use same set `C` for analyzing each branch i in the XOR-block (line 14). This way, we can ensure that activities from a particular branch do not appear in the predecessor sets of activities from another branch of the XOR-block. After every branch is analyzed, we add all activities of this XOR-block to new set `C'` (Line 15).
- **AND-block**. This pattern is analyzed in Lines 17 - 25 of Algorithm 1. Similarly to XOR-blocks, we analyze each branch of an AND-block separately. For every branch i, t_{start} shall point to the first node in branch i and t_{end} to the AND-join node. Further we use same set `C` for every branch i in the AND-block (Line 20). Obviously, an AND-block differs from an XOR-block, since all its branches are executed concurrently. Therefore, for two activities a_i and a_j from two different branches in such AND-block, a_i can appear before a_j in one trace but appear after a_j in another trace. Let B^{AND} represents the AND-block, and B_i^{AND} be the block representing branch i of

B^{AND}. As reflecting on the predecessor set $P(a_i)$ for each $a_i \in B_i^{AND}$, we need to add all activities from other branches $B^{AND} \setminus B_i^{AND}$ to its predecessor set. In Algorithm 1, Line 22 computes set $B^{AND} \setminus B_i^{AND}$ and Line 23 adds them to the predecessors sets $P(a_i)$, $a_i \in B_i^{AND}$. This way, we can ensure that two activities from different branches of an AND-block always appear in the predecessor sets of each other.

- **Loop-block**. This pattern is analyzed in Lines 26 - 29 in Algorithm 1. We first determine which activities are contained in Loop-block B by calling function `parseModel` and adding B to set $\mathcal{L}$ (Line 27). Then, we continue parsing the model using function `parseModel` to analyze the activities inside this Loop-block (Line 28). Note that the analysis of the predecessor sets is based on the simplified trace set (cf. Def. 2), i.e., we only consider the first appearance of trace $t_B \in \mathcal{T}_B$ producible by block B, while later appearances of t_B (caused by the iterative executions of this Loop-block) are not considered any longer. It will handled later by function `addSilentActivitiesForLoopStructure`.

Since blocks may be arbitrarily nested, function `parseModel` in Algorithm 1 is realized as recursive function. We consider sequence structures as basic elements of a process model. Whenever there is an AND- or XOR-block, we consider its branches as blocks and analyze them separately. This division continues until all AND- and XOR-blocks are resolved into blocks which only contains sequence structures with elementary activities. This way we are able to compute predecessor set $P(a_i)$ of each activity a_i in a straightforward way (Lines 7 - 10 in Algorithm 1). Complexity of function `parseModel`, therefore, is $\mathcal{O}(n^2)$, where n equals the number of activities in process models.

As example take process model S in Fig. 1. Table 1 shows analysis results after every step of function `ParseModel` from Algorithm 1. It indicates which node function `parseModel` points to, which activities are processed, which process patterns it belongs to, to which set `C` changes afterwards, and what are the predecessor sets or loop-blocks obtained in this step. For example, Step 1 shows initial state of this function. Steps 2 and 3 analyze the two branches of the AND-block separately, and results are merged in Step 4. After processing activity `D` in Step 5, the algorithm handles the XOR-block in Steps 6-12. Note the differences between Step 4 and Step 12 when dealing with XOR- and AND-joins respectively: additional changes are performed on predecessor sets for activities corresponding to an AND-block. The Loop-block in S, in turn, is handled in Steps 8 - 11, during which we identify which activities are included in this Loop-block.

After obtaining predecessor sets $P(a_i)$ for every activity $a_i \in N$ using function `parseModel`, we can determine order relations between two activities as follows:

- If $((a_i \in P(a_j)) \wedge \neg(a_j \in P(a_i)))$, $A_{a_i a_j}$ = '1', i.e., a_i always precedes a_j.
- If $(\neg\,(a_i \in P(a_j)) \wedge (a_j \in P(a_i)))$, $A_{a_i a_j}$ = '0', i.e., a_i always succeeds a_j.
- If $((a_i \in P(a_j)) \wedge (a_j \in P(a_i)))$, $A_{a_i a_j}$ = '+', i.e., a_i appears before a_j in some traces while it succeeds a_j in other traces.

Step	Node pointing at	Processed activities	Workflow pattern	Parsed activity set C	Predecessor set $P(a_i)$ / Loop-block set B_k
1	AND-split		AND-block	$\emptyset$	
2	A	A	Sequence	$\emptyset$	$P_{(\texttt{A})} = \emptyset$
3	B	B	Sequence	$\emptyset$	$P_{(\texttt{B})} = \emptyset$
4	AND-join	A,B	AND-block	{A,B}	$P_{(\texttt{A})} = \{\texttt{B}\}$ $P_{(\texttt{B})} = \{\texttt{A}\}$
5	D	D	Sequence	{A,B,D}	$P_{(\texttt{D})} = \{\texttt{A},\texttt{B}\}$
6	XOR-split		XOR-block	{A,B,D}	
7	E	E	Sequence	{A,B,D,E}	$P_{(\texttt{E})} = \{\texttt{A},\texttt{B},\texttt{D}\}$
8	Loop-start		Loop-block	{A,B,D}	
9	C	C	Sequence	{A,B,D,C}	$P_{(\texttt{C})} = \{\texttt{A},\texttt{B},\texttt{D}\}$
10	F	F	Sequence	{A,B,D,C,F}	$P_{(\texttt{F})} = \{\texttt{A},\texttt{B},\texttt{D},\texttt{C}\}$
11	Loop-end		Loop-block	{A,B,D,C,F}	$B_k = \{\texttt{C},\texttt{F}\}$
12	XOR-join	E,C,F	XOR-block	{A,B,D,E,C,F}	
13	G	G	Sequence	{A,B,D,E,C,F,G}	$P_{(\texttt{G})} = \{\texttt{A},\texttt{B},\texttt{D},\texttt{E},\texttt{C},\texttt{F}\}$

Table 1. Analysis result for process model S from Fig. 1 when applying `parseModel` in Algorithm 1

– If $(\neg(a_i \in P(a_j)) \wedge \neg\ (a_j \in P(a_i)))$, $A_{a_i a_j}$ = '-', i.e., a_i and a_j never appear together.

The abovementioned method is described by function `computeOrderRelationBasedOnPredecessorSets` in lines 32 - 38 in Algorithm 1. The computation of these four order relations {1,0,+,-} is straightforward since it directly matches with the definition of an order matrix (cf. Def. 3).

As discussed in Section 3, if a process model S contains Loop-blocks, its trace set $\mathcal{T}_S$ becomes infinite. Therefore we need to reduce $\mathcal{T}_S$ to simplified trace set $\mathcal{T}'_S$ for further analysis (cf. Def. 2). We can achieve this reduction by defining one silent activity τ_k for every Loop-block B_k to represent the iterative behavior of the traces producible by B_k (cf. Section 3). After adding activity τ_k to the order matrix, the challenge is to determine order relations between τ_k and the other activities. In the following, we introduce function `addSilentActivitiesForLoopStructure` (Lines 39 - 48 in Algorithm 1) to determine order relations between τ_k and others. In principle, we can divide activities into two groups:

– $a_i \in B_k$. If a_i is contained in the Loop-block, order relation between a_i and τ_k is straightforward. According to Definition 3, $A_{a_i\tau_k} =' L'$ and $A_{\tau_k a_i} =' L'$ must hold.
– $a_i \in N \setminus B_k$. In this case, we need to determine order relation between τ_k and activity a_i which is outside the loop block B_k. Since our process model is block-structured, we can consider whole Loop-block B_k as single "process step" in this context. Therefore, "process step" should have unique order relations with the remaining activities. This implies that all activities belonging to block B_k, including silent activity τ_k, should have same order

relation in respect to the activities located outside this block. We can therefore determine relation between τ_k and $a_i \in N \setminus B_k$ by considering another activity $a_j \in B_k$. Since both τ_k and a_j belong to a same Loop-block, $A_{a_i\tau_k}$ can be assigned to $A'_{a_ia_j}$. Similarly, we obtain $A_{\tau_k a_i} = A'_{a_j a_i}$.

For example take S from Fig. 1. Table 1 depicts predecessor set $P(a_i)$ of each activity a_i, and the set $\mathcal{L}$ for Loop-blocks which we obtain when applying function `parseModel` in Table 1. Based on this, we can compute order matrix A using functions `computeOrderRelationBasedOnPredecessorSets` and `addSilentActivitiesForLoopStructure`. The result is shown in Fig. 1b. Special attention should be paid to the order relations between silent activity τ and the other activities: except the order relations between τ on the one hand and `C` and `F` on the other hand are 'L', the order relations between τ and the remaining activities are same as the ones `C` and `F` have.

Complexity of Algorithm 1 is $\mathcal{O}(2n^2)$ where n equals the number of activities the process model has. To be more precise, the complexity of function `parseModel` is $\mathcal{O}(n^2)$ and complexity of the other two functions corresponds to $\mathcal{O}(n^2)$ in total. This polynomial complexity allows us to quickly transform a (large) process model into its order matrix for further analysis.

5 Transforming an Order Matrix back into a Process Model

In Section 4, we have introduced an algorithm for transforming a process model S into its corresponding order matrix A. In this section, we show how such an order matrix A can be transformed back into a process model S. This approach is described by Algorithm 2.

Algorithm 2 starts with defining a hashtable which maps activities from the order matrix (the key of the hashtable) to their corresponding blocks (the value of the hashtable). Initially, every activity from the order matrix constitutes a block itself (Line 2). The key idea of Algorithm 2 is to merge such blocks. More precisely, two blocks can form a bigger one iff they have same order relations in respect to all remaining blocks within the order matrix (Lines 6 - 9). If two blocks B_i and B_j can be merged into a bigger block B_{ij}, we can build the new block based on these two smaller blocks and their order relation (Lines 11 - 12). The newly created block B_{ij} replaces B_i and B_j in the hashtable. We can then map such block to activity a_i in the order matrix and remove the corresponding row and column for a_j in A (Lines 13 - 15). Therefore, in every iteration, we reduce one row and one column of the order matrix. Merging blocks continues iteratively until there are only two blocks remaining in the order matrix. We merge these two blocks in the last step (Line 23).

Function `createModel`(Block B_i, Block B_j, OrderRelation $\diamond$) creates a process model by merging two blocks B_i and B_j based on their order relation $\diamond$ (Lines 19 - 35 in Algorithm 2). If $\diamond$ represents a predecessor or successor relation, we just need to add one edge between start and end of these two blocks. If the

```
   input  : An order matrix A
   output: A process model S = (N, E, ...)
 1 Define Hashtable Map;
 2 Define each activity a_i as a block B_i; Map.put(a_i, B_i) i = (1, ..., n);
   iteration = 0;                                          /* initial state */ ;
 3 while iteration < n − 2 do
 4     foreach a_i, a_j ∈ N, a_i ≠ a_j do
 5         merge = True ;
 6         for a_k ∈ N \ {a_i, a_j} do
 7             if A_{a_i a_k} ≠ A_{a_j a_k} then
 8                 merge = False;
                       /* two blocks can merge into a bigger one, iff they
                   have same order relations to the others */ ;
 9                 break ;
10         if merge then
11             createModel (Map.getValue(a_i), Map.getValue(a_i), A_{a_i a_j})
               /* create a new block based on the two blocks and their
               order relation */ ;
12             B_ij = buildBlock (Map.getValue(a_i), Map.getValue(a_i), A_{a_i a_j}) ;
                /* Merge these two block based on their order relation */ ;
13             Map.remove(a_i); Map.remove(a_j)       /* change the blocks */ ;
14             A.remove(a_j)                        /* change order relations */ ;
15             Map.put (a_i, B_ij) ;
16             break;
17     iteration ++;
18 S = createModel (Map.getValue(a_1), Map.getValue(a_2), A_{a_1 a_2}) /* Merge last
   two blocks */ ;
19 createModel (Block B_i, Block B_j, OrderRelation ◇)  begin
20     if ◇ = '0' then
           /* Merge block B_i and B_j based on their order relation ◇ */
           addEdge (B_j.end, B_i.start);
21     else if ◇ = '1' then
22         addEdge (B_i.end, B_j.start);
23     else if ◇ = '+' then
24         addNode (AND-split); addNode(AND-join) ;
25         addEdge (AND-split, B_i.start); addEdge (AND-split, B_i.start);
26         addEdge (B_j.end, AND-join); addEdge (B_j.end, AND-join) ;
27     else if ◇ = '-' then
28         addNode (XOR-split); addNode(XOR-join);
29         addEdge (XOR-split, B_i.start); addEdge (XOR-split, B_i.start);
30         addEdge (B_j.end, XOR-join); addEdge (B_j.end, XOR-join) ;
31     else if ◇ = 'l' then
32         Let B_i = τ; addNode (loop-start); addNode(loop-end) ;
33         addEdge (loop-start, B_j.start); addEdge (B_j.end,loop-end);
34         addEdge (loop.end, loop-start, loop) ;
35 end
```

Algorithm 2: Transforming an order matrix into a process model

two blocks have order relation '+' or '-', we add them to different branches of an AND- or XOR-block. If order relation $\diamond$ corresponds to 'L', there must be a Loop-block in the process model and either B_i or B_j correspond a silent activity τ. Reason is that any silent activity τ can only be clustered with Loop-block B that it corresponds to. If B_i equals τ, we only need to surround B_j with a loop-backward edge.

As example take order matrix A from Fig. 1. Since activities `A` and `B` have same order relations in respect to the remaining activities, we are able to merge these two activities to an AND-block. After creating this block, we remove activity `B` from order matrix A. For example, the block containing `A` and `B` can be merged with activity `D` in order to create a bigger block. If we repeat this process of merging blocks, we finally obtain process model S as depicted in Fig. 1.

6 One-to-one Mapping between a Process model and its Order Matrix

Section 4 has introduced Algorithm 1 for transforming a process model into its order matrix. In Section 5, we have further provided Algorithm 2 for transforming an order matrix back into its corresponding process model. Generally, it is critical to prove that such transformation constitutes a one-to-one mapping, i.e., if we first transform a process model S into its order matrix A, and then transform A back to a process model S', S' should be same as S.

When transforming a process model into an order matrix (cf. Algorithm 1), we recursively analyze each block of the process model from start to end. However, when transforming an order matrix back into a process model (cf. Algorithm 2), the order in which we merge blocks is more or less arbitrary, i.e., we merge two blocks together whenever this is possible. Therefore, it is important to know whether the order relation satisfies the *associative law*, i.e., whether the order to merge small blocks into bigger ones can influence the result.

Let us first consider predecessor-successor order relations '0' and '1'. Assume process model S contains three blocks B_i, B_j and B_k with B_i preceding B_j and B_j preceding B_k. Obviously, we obtain $A_{B_iB_j} =' 1'$ and $A_{B_jB_k} =' 1'$. Representing this as mathematical equation, we obtain $B_i \diamond B_j \diamond B_k$ with $\diamond$ being '1'. If we need to transform order matrix A into a process model S', it does not matter whether we first merge B_i and B_j or we first merge B_j and B_k, i.e., $(B_i \diamond B_j) \diamond B_k = B_i \diamond (B_j \diamond B_k)$. It is obvious from Algorithm 2 that we only need to add control edges between the end of the preceding block and the start of the succeeding one. Therefore the order of adding edges does not influence the resulting model. Clearly, such rule also applies if order relation $\diamond$ corresponds to '0'.

For order relations '+' and '-', let us re-assume that there are three blocks B_i, B_j and B_k with same order relation '-' among each other, i.e., $B_i \diamond B_j \diamond B_k$ with $\diamond$ being '-'. Fig. 2a shows two models S and S' that result when merging these three blocks together. To be more precise, S can be obtained by first merging B_i and B_j and then merging the intermediate block with B_k ($S = (B_i \diamond B_j) \diamond B_k$),

while S' can be obtained by first merging B_j and B_k and then merging the resulting block with B_i ($S' = B_i \diamond (B_j \diamond B_k)$).

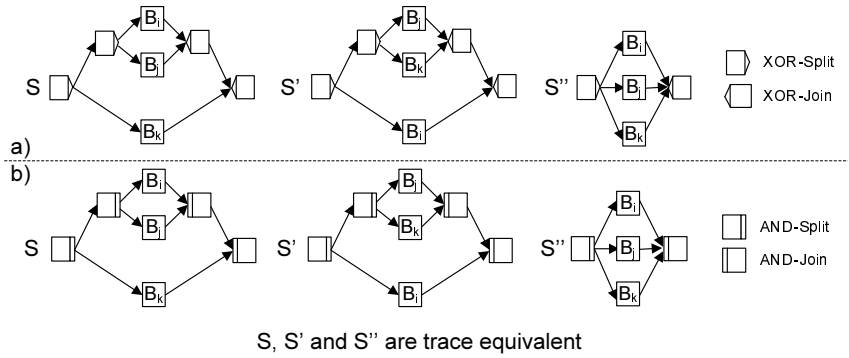


Fig. 2. Trace equivalence for AND and XOR blocks

Obviously, process models S and S' are structurally different. However, S and S' are trace equivalent despite their structural difference, i.e., trace sets $\mathcal{T}_S$ and $\mathcal{T}_{S'}$ producible on S and S' respectively are the same: $\mathcal{T}_S = \mathcal{T}_{S'} = \mathcal{T}_{B_i} \bigcup \mathcal{T}_{B_j} \bigcup \mathcal{T}_{B_k}$. Note that the two process models are trace equivalent to S'' as well. Regarding S'', B_i, B_j and B_k are located in three different branches of the same XOR-block. In the context of our research, we consider S, S' and S'' being the same, since the corresponding process models have same trace set and therefore show same behaviors.[8] This indicates that order relation '-' satisfies the *associative law*, and consequently the order in which blocks are merged is not relevant. When first transforming a process model (e.g., S'') into order matrix A, and then transforming A back into a process model (e.g., S'), the original and the newly derived process models are trace equivalent. In our context, we consider these two models being same. Thus the transformation between process model and order matrix constitutes a one-to-one mapping. Obviously, same results are obtained if the order relations between them are '+' (cf. Fig. 2b).

For order relation 'L', the associative law is not applicable in the given context because $B_i \diamond B_j \diamond B_k$ (with $\diamond$ = 'L') is not possible for an order matrix. If $B_i \diamond B_j$ holds with $\diamond$ = 'L', either B_i or B_j must be τ. This, in turn, indicates that in expression $S_i \diamond S_j \diamond S_k$, two out of the three blocks constitute silent activities τ. Let us assume that B_j and B_k are these two silent activities. Thus such expression means that block B_i is surrounded by a loop-backward edge to form a loop-block B_i' and this Loop-block B_i' is immediately surrounded by another loop-backward edge to form another Loop-block B_i'', i.e., B_i is surrounded by two loop-backward edges. In this case, B_i' and B_i'' are trace equivalent, and

[8] Like most process mining techniques (e.g. [30, 7, 34]), the stronger notion of bi-similarity are not considered in our context [9]

therefore expression $S_i \diamond S_j \diamond S_k$ would be simplified to $S_i \diamond S_j$ with $\diamond$ being 'L'.[9] This implies that the condition to analyze the associative law does not hold.

Although associative law is not applicable to order relation 'L', it cannot influence the one-to-one mapping between a process model and its order matrix. Reason is that whenever two blocks have order relation 'L', one of them must be a silent activity τ, and this silent activity can only be merged with the block surrounded by a loop-back edge. Assume that block B is surrounded by a loop, i.e., has order relation 'L' with silent activity τ (cf. Def. 3). For any block B_i which is different from B, there are only two situations: either B contains activities not in B_i or B_i is a sub-block of B. In the first scenario, for any activity $a_i \notin B$, a_i must have different order relation in respect to τ than activities in B have, therefore B_i cannot be clustered with τ; in the second scenario, there must be an activity $a_i \in B \setminus B_i$ having different order relation to τ when compared to an activity $a_j \in B_i$. Therefore B_i can not be clustered with τ. This indicates that the silent activity τ can only be clustered with the Loop-block it corresponds to. Consequently, the order of clustering also does not influence results.

7 Algebraic Properties of Order Relations

In addition to associativity of order relations, we have analyzed their algebraic properties. Let $S_i, S_j, S_k \in \mathcal{P}$ be three sound process models. In this context, we denote a process model as sound if there are no deadlocks or unreachable activities in the process model [21, 28]. Let further $\diamond = \{0, 1, +, -, L\}$ be the order relations as set out by Definition 3. Then, the algebraic system $< \mathcal{P}, \diamond >$ has the properties depicted in Table 2:

Algebraic property		Order relation $\diamond$				
Name	Mathematical expression	0	1	+	-	L
Closure	$S_i \diamond S_j \in \mathcal{P}$	Yes	Yes	Yes	Yes	Yes
Commutativity	$S_i \diamond S_j = S_j \diamond S_i$	No	No	Yes	Yes	Yes
Transitivity	$S_i \diamond S_j \wedge S_j \diamond S_k \Rightarrow S_i \diamond S_k$	Yes	Yes	No	No	Yes
Associativity	$(S_i \diamond S_j) \diamond S_k = S_i \diamond (S_j \diamond S_k)$	Yes	Yes	Yes	Yes	- [10]
Identity element I	$S_i \diamond I = S_i$	$I = \emptyset$	$I = \emptyset$	$I = \emptyset$	None	$I = \emptyset$ [11]

Table 2. The algebraic properties of order relation

[9] We can easily identify such situation from the process model or the order matrix. If a block is surrounded by two loop-back edges in a process model, we only need to keep one of them so that the model is still trace equivalent to the original one. In an order matrix, we can easily identify such situation by checking whether two silent activities τ is able to merge or not. If yes, then we can remove one of them.

[9] According to the analysis in Section 6, conditions for analyzing associative law does not exist

[11] If two blocks S_i and S_j have order relation 'L', at least one of them must be silent activity τ. Therefore, we consider the identity element also being existent for order

In details, Table 2 summarizes 5 algebraic properties of the order matrix, namely, closure, commutativity, transitivity, associativity and identity element. The analysis is based on function `createModel` as defined in Algorithm 2. Using this function we can merge two process models S_i and S_j (a process model is also a block) into another process model S_{ij} based on order relation $\diamond$.

- **Closure**. For all five order relations, the closure property is satisfied, i.e., if we merge two sound process model S_i and S_j based on any of the five order relations $\diamond = \{0, 1, +, -, L\}$, we obtain another sound process model S_{ij}. This is a very important property since it guarantees soundness of the resulting model when merging two process models using function `createModel` (cf. Algorithm 2). Consequently, when transforming an order matrix into a process model using Algorithm 2, the resulting process model must be sound as well, since the algorithm constructs a process model by merging blocks (cf. Algorithm 2). The theoretical background to guarantee soundness of the resulting model when merging two blocks can also be found in ADEPT change framework [21] (or see also inheritance rule in Petri Nets [29]).
- **Commutativity**. Commutativity is represented by the relation of the elements in an order matrix. If $A_{a_i a_j} = \diamond$ with $\diamond \in \{+,-,L\}$ holds, we will obtain $A_{a_j a_i} = A_{a_i a_j}$ (since the respective order relations satisfy commutative law). On the contrary, if $A_{a_i a_j} =$ '0' holds, we can obtain $A_{a_j a_i} =$ '1' and vice verse. This implies that, in most cases it is sufficient to only analyze the upper or lower part of the triangle in the order matrix.
- **Transitivity**. Transitive law applies to order relations '0', '1' and 'L', but not to the others. This property is important for Algorithm 1 because the key construct in this algorithm is a sequence structure with start- and end-node. Since order relations '0' and '1' are transitive, but not commutative, for a given process model S with finite number of activities, there must be a start- and end-node. Reason is that if an algebraic system with finite number of elements is transitive, but not commutative, this algebraic system must be bounded [25], i.e., there must be an element which does not have predecessors (the start of process model) and an element which does not have any successor (the end of process model).
- **Associativity**. The associative law has been discussed in Section 6. This property guarantees that when transforming a process model S into an order matrix A, and then transforming resulting order matrix back to a process model S', S and S' are trace equivalent. This property is important in order to guarantee the one-to-one mapping between a process model and its order matrix.
- **Identity element**. The identity element is important for dealing with silent activities in a process model. Since silent activities constitute the identity element for order relations '0', '1', '+' and 'L', we do not need special treatment for them (i.e., they will not influence the result). This property is important,

relation 'L', since an "empty" process model remains an "empty" process model, even after surrounding it within a loop structure.

especially when changing a process model and its corresponding order matrix. Note that such changes often introduce silent activities [13, 29, 21, 32]. The only exception is provided by order relation '-'. When merging a process model S_i with a silent activity based on order relation '-', we obtain a new block S_j. Consequently, we need to add one more empty trace t (producible by the silent activity, i.e., a trace contains no activity) into trace set $\mathcal{T}_{S_i}$ in order to obtain trace set $\mathcal{T}_{S_j}$. Therefore, if $t \notin \mathcal{T}_{S_i}$, we obtain $\mathcal{T}_{S_i} \neq \mathcal{T}_{S_j}$. We refer to [13] for an approach to handle silent activities for order relation '-'.

8 Conclusion

This paper provides a matrix representation of a process model, which we denote as order matrix. We have presented an algorithm to transform a process model into its order matrix, and an algorithm to transform an order matrix back to a process model. We have further shown that the mapping between process model and its order matrix is one-to-one, i.e., we basically obtain same process model when transforming a process model into an order matrix and then transforming the resulting order matrix back to a process model. Finally, we have discussed algebraic properties of order relations and their influences on the algorithms as well.

References

1. R. A. Baeza-Yates. Text-retrieval: Theory and practice. In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1*, pages 465–476, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.
2. R. P. Jagadeesh Chandra Bose and W. M. P. van der Aalst. Context aware trace clustering: Towards improving process mining results. In *SDM'09*, pages 401–412. SIAM, 2009.
3. BPEL. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.
4. OMG BPMI.org. *Business Process Modeling Notation Specification.* Object Management Group, 2006. available at: http://www.bpmn.org.
5. P. F. Brown, V. J. Della Pietra, P. V. de Souza, J. C. Lai, and R. L. Mercer. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479, 1992.
6. P. Dadam and M. Reichert. The ADEPT project: A decade of research and development for robust and flexible process support - challenges and achievements. *Computer Science - Research and Development*, 23(2):81–97, 2009.
7. A.K. Alves de Medeiros. *Genetic Process Mining.* PhD thesis, Eindhoven University of Technology, NL, 2006.
8. A. Hallerbach, T. Bauer, and M. Reichert. Capturing variability in business process models: the provop approach. *Software Process: Improvement and Practice*, page to appear, 2009.
9. J. Hidders, M. Dumas, W.M.P. van der Aalst, A.H.M. ter Hofstede, and J. Verelst. When are two workflows the same? In *CATS '05:*, pages 3–11, Darlinghurst, Australia, 2005.

10. B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler. On structured workflow modelling. In *CAiSE'00*, pages 431–445. LNCS 1789, Springer, 2000.
11. C. Li, M. Reichert, and A. Wombacher. Discovering reference process models by mining process variants. In *ICWS'08*, pages 45–53. IEEE Computer Society, 2008.
12. C. Li, M. Reichert, and A. Wombacher. Mining process variants: Goals and issues. In *IEEE SCC (2)*, pages 573–576. IEEE Computer Society, 2008.
13. C. Li, M. Reichert, and A. Wombacher. On measuring process model similarity based on high-level change operations. In *ER '08*, pages 248–262. Springer LNCS 5231, 2008.
14. C. Li, M. Reichert, and A. Wombacher. Discovering reference models by mining process variants using a heuristic approach. In *BPM'09, LNCS 5701*, pages 344–362. Springer, 2009.
15. J. Mendling. *Metrics for Process Models: Empirical Foundations of Verification, Error Prediction and Guidelines for Correctness*, volume 6 of *LNBIP*. Springer, 2008.
16. J. Mendling, G. Neumann, and W. M. P. van der Aalst. Understanding the occurrence of errors in process models based on metrics. In *CoopIS'07, LNCS 4803*, pages 113–130, 2007.
17. J. Mendling, H. A. Reijers, and J. Cardoso. What makes process models understandable? In *BPM'07*, pages 48–63. LNCS 4714, Springer, 2007.
18. J. Mendling, H. A. Reijers, and W. M. P. van der Aalst. Seven process modeling guidelines (7pmg). *Information & Software Technology*, 52(2):127–136, 2010.
19. J. Mendling, B. F. van Dongen, and W. M. P. van der Aalst. Getting rid of or-joins and multiple start events in business process models. *Enterprise IS*, 2(4):403–419, 2008.
20. J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., USA, 1993.
21. M. Reichert and P. Dadam. ADEPTflex -supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
22. M. Reichert, S. Rinderle-Ma, and P. Dadam. Flexibility in process-aware information systems. *LNCS Transactions Petri Nets and Other Models of Concurrency*, 2:115–135, 2009.
23. H. A. Reijers and J. Mendling. Modularity in process models: Review and effects. In *BPM'08*, pages 20–35. LNCS 5240, Springer, 2008.
24. S. Rinderle-Ma, M. Reichert, and B. Weber. On the formal semantics of change patterns in process-aware information systems. In *ER'08*, pages 279–293. LNCS 5231, Springer, 2008.
25. K.H. Rosen. *Discrete Mathematics and Its Application*. McGraw-Hill, 2003.
26. P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.
27. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
28. W. M. P. van der aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. The MIT Press, 2002.
29. W.M.P. van der Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theor. Comput. Sci.*, 270(1-2):125–203, 2002.
30. W.M.P van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1128–1142, 2004.

31. J. Vanhatalo, H. Volzer, and J. Koehler. The refined process structure tree. *Data Knowledge Engineering*, 68(9):793–818, 2009.
32. B. Weber, M. Reichert, and S. Rinderle-Ma. Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data and Knowledge Engineering*, 66(3):438–466, 2008.
33. B. Weber, S. Wasim Sadiq, and M. Reichert. Beyond rigidity - dynamic process lifecycle support. *Computer Science - R&D*, 23(2):47–65, 2009.
34. A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering workflow models from event-based data using little thumb. *Integr. Comput.-Aided Eng.*, 10(2):151–162, 2003.
35. A. Wombacher and M. Rozie. Evaluation of workflow similarity measures in service discovery. *Service Oriented Electronic Commerce*, pages 51–71, 2006.
36. M. zur Muehlen and J. Recker. How much language is enough? theoretical and practical use of the business process modeling notation. In *CAiSE'08*, pages 465–479. LNCS 5074, Springer, 2008.